2016-06

Empirical analysis of using erasure coding in outsourcing data storage with provable security

Bakir, Abdallah

Monterey, California: Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

EMPIRICAL ANALYSIS OF USING ERASURE CODING IN OUTSOURCING DATA STORAGE WITH PROVABLE SECURITY

by

Abdallah Bakir

June 2016

Thesis Advisor:	Mark Gondree
Second Reader:	Raymond R. Buettner

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	06-17-2016	Master's Thesis 09-01-2015 to 06-17-2016

4. TITLE AND SUBTITLE

EMPIRICAL ANALYSIS OF USING ERASURE CODING IN OUTSOURCING DATA STORAGE WITH PROVABLE SECURITY

5. FUNDING NUMBERS

6. AUTHOR(S)

Abdallah Bakir

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Proof of retrievability (POR) and proof of data possession (PDP) are cryptographic tools for auditing big data on a storage server or in the cloud. Their goals are to verify that the server is storing data and, in case of data alteration, recovering this data. These tools provide probabilistic guarantees that the server is storing information, without accessing the entire file and providing the capability to recover the original data under certain limits. In this work, we study maximum distance separable (MDS) codes as the underlying tools providing recoverability for POR. We survey MDS codes and select Reed-Solomon and Cauchy Reed-Solomon MDS codes to be implemented into a prototype POR library. We use the *liberasurecode* library to evaluate multiple error-correcting code (ECC) backend implementations for these codes. We enhance the *libpdp* library, an open source PDP library that implements some PDP schemes, to interface with *liberasurecode* to measure the real-world cost of integrating erasure coding in POR implementations.

14. SUBJECT TERMS

Proof of retrievability, proof of data possession, Erasure codes, error correcting code, cloud storage, data integrity,

15. NUMBER OF PAGES 53

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

EMPIRICAL ANALYSIS OF USING ERASURE CODING IN OUTSOURCING DATA STORAGE WITH PROVABLE SECURITY

Abdallah Bakir
First Lieutenant, Tunisian Army
B.S.,Tunisian Military Academy, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION WARFARE SYSTEMS ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2016

Approved by: Mark Gondree
Thesis Advisor

Raymond R. Buettner
Second Reader

Dan Boger
Chair, Department of Information Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Proof of retrievability (POR) and proof of data possession (PDP) are cryptographic tools for auditing big data on a storage server or in the cloud. Their goals are to verify that the server is storing data and, in case of data alteration, recovering this data. These tools provide probabilistic guarantees that the server is storing information, without accessing the entire file and providing the capability to recover the original data under certain limits. In this work, we study maximum distance separable (MDS) codes as the underlying tools providing recoverability for POR. We survey MDS codes and select Reed-Solomon and Cauchy Reed-Solomon MDS codes to be implemented into a prototype POR library. We use the *liberasurecode* library to evaluate multiple error-correcting code (ECC) backend implementations for these codes. We enhance the *libpdp* library, an open source PDP library that implements some PDP schemes, to interface with *liberasurecode* to measure the real-world cost of integrating erasure coding in POR implementations.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AWS	Amazon Web Services
CRS	Cauchy Reed-Solomon
DOD	Department of Defense
ECC	error-correcting code
EC2	elastic compute cloud
HDFS	Hadoop Distributed File System
I/O	input/output
MAC	message authentication code
MDS	maximum distance separable
NC	network coding
NPS	Naval Postgraduate School
PDP	proof of data possession
POR	proof of retrievability
RAID	redundant array of independent disks
RS	Reed-Solomon
USG	United States government

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to express my gratitude and appreciation to my advisor, Professor Mark Gondree, for his direction and guidance in helping me to develop and write my thesis. Professor Gondree was willing to provide advice whenever I ran into a trouble spot or had a question about my research or writing.

I would especially like to thank my wife, Rabeb, and my daughter Farah Eljannah. My wife devoted herself to support and encourage me throughout this Thesis. Also, my daughter is my source of inspiration and delight, and she provided me with the energy to fulfill this thesis.

Last but not the least, I would like to thank all my family: my parents, my brother, and my sisters, for supporting me spiritually throughout the writing of this thesis and my life in general.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Both proof of data possession (PDP) and proof of retrievability (POR) are cryptographic protocols for auditing data stored in remote servers without accessing an entire file. PDP and POR are like a digital signature but are efficient for periodic audits of massive data sets, at the cost of weaker, probabilistic integrity guarantees.

Auditing cloud storage is an important issue that many scholars have studied. While different approaches have been suggested, no open source solutions exist providing the features or ease-of-use to inspire wide adoption in auditing and recovering data held by cloud storage. Indeed, further study is warranted in investigating the practicality of POR toward the goal of an open source cloud storage audit solution.

Few analyses have been performed comparing the cost of retrievability for POR schemes. Proposed POR schemes employ some error-correcting code (ECC) mechanism, which imposes an additional encoding cost, storage overhead, and decoding costs during data tagging, recovery and update.

In this thesis, we implement and evaluate POR schemes supporting recovery based on maximum distance separable (MDS) codes. We explore the real-world costs of MDS codes, to understand consequential issues for making POR practical in the single-server model. We focus on encoding, storage and tagging costs. We defer exploring decoding and update cost to future work.

Our work makes the following primary contributions:

- We enhance an open source PDP library, *libpdp*, to provide retrieval guarantees;
- We explore the costs associated with POR from practical, rather than asymptotic, perspectives.
- As expected, we find the cost of tagging to be proportional to the increase in the file size resulting from the use of the MDS code and closely follows the performance costs reported by Bremer [1].
- We find that the parity data add additional storage cost and the ratio m/k defines the

overhead amount.

1.1 Motivation

In today's Information age, where computing and communication technologies become powerful and advanced, people are exchanging a huge amount of data, and they are demanding more storage capability for personal, business, or even military needs. The International Data Corporation's sixth annual study of the digital universe [2] showed that the digital universe data is growing by 40 percent a year. They stated also that the digital data was about 4.4 ZB in 2013 and it is expected to reach 44 ZB by 2020. Furthermore, users require access to their data even when they are away from their personal computer or portable storage devices. Today's users expect on-demand access to their data anywhere in the world. The cloud is a suitable solution to provide the benefits of high storage capability, greater accessibility, reliability, archival, and data backup. For instance, the same IDC report states that, by 2020, almost 40% of the digital universe data will be "touched" by cloud computing providers. Also, in 2012, DOD Chief Information Officer Teresa M. Takai released the Department of Defense Cloud Computing Strategy in an effort to cope with the massive growth of data and to reduce data storage costs.

Yet the cloud presents another issue related to data integrity. It is not certain if the user can rely entirely on the cloud server to manage his data. Also, it is impossible to be sure of the server's behavior. Consequently, the security of stored data is one of the big concerns for organizations and individuals. For instance, a 2015 Vormetric data security study [3] stated that 60% of IT decision makers report keeping sensitive data locally, not in the cloud. Their concerns are related to lack of data control and increased vulnerabilities.

1.2 Outline

In Chapter 2, we provide background on proof of retrievability approaches and error-correcting codes. In Chapter 3, we describe a specific type of ECC that is capable of correcting adversarial erasures and survey codes satisfying this requirement. In Chapter 4, we discuss design criteria for using MDS codes in POR schemes. In Chapter 5, we discuss a prototype POR library and evaluate the performance related to the aspects introduced to support retrievability. In Chapter 6, we conclude and discuss future work.

CHAPTER 2:
Background

In this chapter, we review proof of retrievability (POR) schemes, the single server and distributed failure models, and the role of error-correcting codes (ECCs) in retrieving data.

2.1 Proof of Retrievability

POR enables a client to remotely audit stored data in a manner robust against adversarial erasures, at a cost substantially less than traditional digital signatures. This is achieved through an interactive challenge-response protocol implementing a random, sublinear audit of file integrity. Assuming some ϵ-fraction of audits is passed, POR schemes admit an efficient mechanism to extract and recover the original file. A summary of major POR schemes and their properties is provided in Table 2.1.

Juels and Kaliski [4] and Naor and Rothblum [5] proposed the first approaches to remotely audit cloud storage at sublinear cost. Their framework was based on splitting the original file into chunks, signing chunks using a cryptographic message authentication code (MAC) scheme to generate a set of data tags, and storing both the data chunks and tags remotely. The file could then be remotely audited using an interactive protocol, where a random subset of chunks and their corresponding tags could be retrieved and verified. This random sublinear audit could be repeated, arbitrarily increasing the client's confidence in the integrity of the data without retrieving the entire file.

A generalization of these early schemes was later shown to admit efficient mechanisms for file recovery, yielding a POR scheme [6]. Ateniese et al. [7] and Shacham and Waters [6] proposed schemes with lower communication cost, also shown to admit file recovery under appropriate pre-processing steps. These schemes are all in the *single-server failure model*. In this model, all data is held by a single remote cloud service and data loss occurs due to adversarial block-level erasures. File pre-processing uses an appropriate ECC to achieve recovery against adversarial erasure (see Chapter 3).

Later schemes considered a different failure model, the *multiple-server failure model*. In this model, all data is distributed across multiple servers and data loss occurs due to

adversarial file deletion at the server level. Three broad approaches exist to add redundancy in distributed storage systems: file replication, network coding (NC) and error-correcting codes (ECCs). POR recovery mechanisms have been based on each of these approaches. Curtmola et al. [8] provide a POR scheme ensuring that multiple unique copies of the data are stored across different servers. Chen et al. [9] propose a POR scheme for network coding-based distributed storage. Bowers et al. [10] introduce a POR scheme, which permits a set of servers to prove to a client that data is stored and retrievable, employing error correction codes.

Table 2.1: Comparison of existing POR schemes.

Model	Scheme	Year	Recovery approach	Storage complexity	Communication complexity
Single Server	MAC-PDP [4], [5]	2007	ECC	$O(\lvert\widetilde{F}\rvert + n\lvert\sigma\rvert)$	$O(1)$
	A-PDP [7]	2007	ECC	$O(\lvert\widetilde{F}\rvert/k)$	$O(1)$
	CPOR [6]	2008	ECC	$O(\lvert\widetilde{F}\rvert/k + n\lvert\sigma\rvert)$	$O(1)$
Multiple Server	MR-PDP [8]	2008	replication	$O(\lvert S\rvert\lvert F\rvert)$	$O(\lvert F\rvert)$
	HAIL [10]	2009	ECC	$O(\lvert S\rvert\lvert F\rvert/k)$	$O(\lvert F\rvert)$
	RDC-NC [9]	2010	NC	$O(2\lvert S\rvert\lvert F\rvert/(k+1))$	$O(2\lvert F\rvert/(k+1))$

See Section 2.3 for relevant notation. For multiple server schemes, $|S|$ denotes the number of servers employed; for HAIL, $|S| = k + m$ where k primary servers store the original fragments and m secondary servers store parity fragments. Adapted from [9, Table 1]: B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, 2010, pp. 31–42.

2.2 Error Correction Codes

Error detection and error correction are mathematical procedures enabling reliable storage and transmission of data, providing a way to detect and correct errors caused by channel noise. The idea is to transform the data while incorporating redundancy, allowing the original data to be recovered even in the presence of some limited transmission errors (see Figure 2.1). Shannon [11] first proposed a mathematical model for data transmission across a channel in the presence of noise, effectively defining the field of *information theory*. In this

model, Shannon proved a theoretic limit (the noisy-channel coding theorem) on the maximum efficiency of *any* error correction strategy under some rate of stochastic/probabilistic transmission errors. Hamming [12] provided the first concrete ECC to achieve this optimal rate with minimal distance between codewords, called a *perfect code*. Hamming's code used a combinatorial argument to ensure unique decoding, rather than an argument based on the likelihood of correlated errors, thus allowing *arbitrary* error detection up to some threshold.

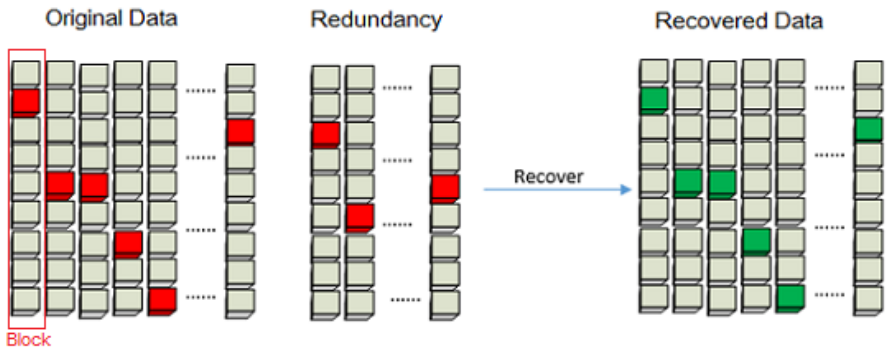


Figure 2.1: Error recovery using ECC.

Adapted from [13]: J. S. Plank and C. Huang, "Tutorial: Erasure coding for storage applications," Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, San Jose, February 2013.

Broadly, ECCs are divided into two categories: block codes and convolution codes. Block codes divide data into blocks, called *datawords*, and transform each dataword into a *codeword*. In convolution codes, redundancy not only depends on the current inputs bits but also some amount of the previous bits.

For ECCs, the two most common channel error models are independent errors and burst errors. Independent, or random, errors are those produced by a memoryless, stochastic noise process: codewords are impacted by noise according to some constant probability, with no correlation in time or location among errors. Burst errors are those where noise is localized in short intervals rather than at random. For example, the Gilbert-Elliot burst error model uses a two-state Markov process to switch, with some probability, between "good" and "burst" states, incurring transmission errors in the burst state with some rate. In the real world, burst errors are intended to describe physical processes with correlated

noise; for example, errors resulting from physical irregularities in storage media or structural alteration are likely not independent but, rather, tend to be spatially concentrated.

The type of error model expected is crucial to selecting a suitable error correction code. Codes that are good at correcting random errors may do a poor job at correcting burst errors, or vice-versa. Other error models have been proposed, including the adversarial error model (discussed further in Chapter 3) and the computationally bounded error model [14], [15]. Both consider more general failure scenarios, where deletions may be performed by an active adversary, selecting noise following some adversarial strategy rather than at random.

2.3 Notation

In this work, we consider the properties of POR schemes in the single-server failure model. For these, a client C transforms a file F into an encoded file $\widetilde{F}$. The client stores this data with some cryptographic metadata (tags) using some remote storage service S. Later, the client or a third-party auditor will remotely verify the integrity of the file data through an interactive protocol. Assuming some ϵ-fraction of audits succeed, it should be possible to extract the original file F. In the remainder of this thesis, we employ the following notation whenever describing these schemes:

- F The original file of size $|F|$.
- k An encoding parameter, used to divide F into k data blocks.
- μ The data block size, $\mu = \lceil |F|/k \rceil$.
- w The data block is considered to be a series of w-bit words.
- m An encoding parameter, used to extend the k data blocks into m parity blocks.
- $\widetilde{F}$ The encoded file, to be tagged and remotely stored.
- σ A single tag of size $|\sigma|$.
- n An audit parameter, the total number of chunks in $\widetilde{F}$.
- λ An audit parameter, the number of chunks to be queried on each audit.

Some encoding schemes may divide F into L logical blocks of size ℓ, encoding each logical block into k data blocks and m parity blocks. Some audit schemes may further divide audit chunks into sectors. The previously identified parameters will be the most useful and common notation for our discussion.

CHAPTER 3:
Adversarial Erasure Coding

In this chapter, we define and review adversarial erasure codes. We survey maximum distance separable codes, which are able to correct worst-case errors like those considered by Hamming [12] and are one category of adversarial erasure code.

3.1 Definition

Hamming's *adversarial channel* [12] introduces noise in a worst-case manner, flipping or erasing bits arbitrarily according to some computationally unbounded strategy up to some error rate. Lipton [14] proposes a variant of this model in which the adversarial strategy is *computationally bounded*, modeled by an arbitrary polynomial-time Turing machine. Shacham and Waters [6] give a construction for an adversarial erasure code able to correct worst-case errors placed following a computationally bounded strategy, using an efficient linear-time erasure code, scrambled and protected using encryption; this essentially causes any computationally bounded strategy to have no better success than random erasures. Bowers et al. provide a related construction for a systematic, adversarial error correcting code.

3.2 Maximum Distance Separable Codes

The Singleton bound describes the possible relationship between codeword length and the distance between codewords. For a linear code with c codewords, each of size M and minimum distance d, the bound is typically expressed as

$$d \leq c - M + 1.$$

MDS codes are linear codes in which the minimum distance between any two codewords meets the Singleton bound with equality [16]. These have been proposed for the fault tolerance in many applications, including cloud storage. For example, Bowers et al. propose maximum distance separable codes to ensure retrievability of data stored in distributed

storage [10].

MDS codes divide a file F into k blocks of size $\frac{|F|}{k}$. These k blocks are used to calculate m parity fragments. The k blocks and m parity blocks are concatenated together to form n data blocks, which can be considered an array of w-bit codewords. If the encoded file is altered, the original file can be recovered from any k of the n encoded fragments. The parameters k and m are parameters that depend on the type of code and user requirements.

Shacham and Waters observe that erasure codes derived from Reed-Solomon codes have the property that they can decode in the presence of adversarial erasures [6, §1.1]. The property desired is exactly those of MDS codes, able to correct up to m adversarial erasures. These codes are traditionally criticized for the reasons that (i) they are inefficient and (ii) their goals are unrealistically strong, i.e., correcting noise placed by an adversary that is computationally unbounded. More efficient codes are possible and discussed by Shacham and Waters [6] and Bowers et al. [17]. We evaluate MDS codes as a first step toward the future implementation of libraries supporting more efficient adversarial erasure codes.

3.3 MDS Survey

In this section, we survey MDS codes, comparing those parameters that appear consequential to storage and retrievability for POR schemes. Different ECCs have different structures and aim to satisfy specific goals: some seek to increase recovery efficiency, others focus on minimizing recovery costs, etc. We focus on the following properties:

Coding chunks. A bound on m, the number of blocks reserved for parity.

Storage overhead. The difference in the amount of stored data between an original file and the encoded file.

Fault tolerance. The maximum number of blocks that may be lost while allowing recovery of the original file.

Cost of retrievability. The number of fragments needed to recover from one failure.

Results are summarized in Table 3.1. For details on the MDS codes summarized here, we refer the reader to the existing survey work of Plank et al. [18] and Schnjakin et al. [19].

EVENODD and Minimal Density codes have limits on the number of coding chunks and, therefore, limits their fault tolerance. Reed-Solomon (RS), Cauchy Reed-Solomon (CRS), and Rotated RS are more flexible, placing no bounds on the number of coding chunks that may be added.

Table 3.1: Comparison of MDS code properties.

Code	Coding Chunks	Storage Overhead	Fault Tolerance	Cost of Retrievability
RS [20]	∞	n/k	$n-k$	$n\frac{\mu}{w}$
CRS [21]	∞	n/k	$n-k$	$< n\frac{\mu}{w}$
EVENODD [22]	2	$1+2/k$	2	$< n\frac{\mu}{w}$
Minimal Density [23]	2	$1+2/k$	2	$< n\frac{\mu}{w}$
Rotated RS [24]	∞	n/k	$n-k$	$\frac{r}{2}(n+\frac{n}{k})\frac{\mu}{w}$

Adapted from [25]: M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2010, pp. 66–75.

For recovery, RS codes are considered expensive due to multiplication operations during decoding. While their asymptotic costs appear similar, CRS are considered to provide more acceptable recovery cost. They reduce of the number of operations by using Cauchy matrices (which have fewer ones than the Vandermonde matrices used RS codes) and XOR operations (rather than multiplication). Rotated RS codes transform chunks of w-bit size words into r groups of bit-words. These have improved recovery performance since they are designed to provide better input/output efficiency: accessing data at the bit-level requires less data to be read during recovery.

In recent years, several open-source implementations of MDS codes have been made available. In Table 3.2, we update earlier survey work of Plank et al. [18] and Schnjakin et al. [19], excising libraries that are no longer available and including new libraries that have been developed. Most libraries are written in C/C++, the exception being *JavaReedSolomon*. Intel's *ISA-L* library provides raw interfaces to take advantage of hardware acceleration for operations used in erasure coding, as well as providing direct implementation of some codes [26]. The library *liberasurecode* supports multiple ECC backends, including *Jerasure* and *ISA-L*. Almost all libraries support RS, CRS or both, each providing different implementations and optimizations.

Table 3.2: Survey of MDS support in open source ECC libraries.

Library	RS	CRS	EVENODD	Minimal Density	Rotated RS
Intel ISA-L [26]	✓	✓	-	-	-
JavaReedSolomon [27]	✓	-	-	-	-
Jerasure [28]	✓	✓	-	✓	-
Kodo [29]	✓	-	-	-	-
liberasurecode [30]	✓	✓	-	✓	-
Luby [21]	-	✓	-	-	-
OpenFEC [31]	✓	-	-	-	-
PyECLib [32]	✓	✓	-	✓	-
Zfec [33]	✓	-	-	-	-

Adapted from [25]: M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2010, pp. 66–75.

Many of these implementations are used in practical storage solutions or made accessible through other libraries. The library *PyECLib* is a python wrapper for *liberasurecode* [32], developed for use in Swift [34]. The library *librain* is a wrapper for *Jerasure* used by the OpenIO software defined storage project [35]. As of Hadoop Distributed File System 3.0, the HDFS project has incorporated its own Java implementations of XOR and RS codes, taking advantage of the *ISA-L* library for hardware acceleration [36], [37], the goal of incorporating ECC with HDFS having been previously explored by Facebook's HDFS-RAID project [38].

Plank et al. [39] provide a practical-oriented performance analysis of many available open-source ECC libraries. Their evaluation shows that *Zfec* implements the fastest classic RS coding, with *Jerasure* showing the best implementation among the other schemes evaluated. They highlight some parameters that impact performance in each library. Almost all implementations evaluated are sensitive to the codeword size, w. In particular, $w \in \{8, 12, 13, 14, 16, 19, 24, 26, 27, 30, 32\}$ yields poor performance for CRS, since the primitive polynomial for these fields have one more bit set than in other cases. Plank et al. also reflect on the role memory and cache footprints play in implementation efficiency; in particular, they attribute the better performance of *Zfec* to its smaller memory footprint.

The *EVEN/ODD* code is not an adequate solution for practical adversarial error correction since it has limits on the number of errors that can be corrected, and there are no publicly available implementations. Following the same rationale, *Minimal Density* will not satisfy our needs. Schnjakin et al. conclude that *Rotated RS* is not beneficial in cloud storage applications because most cloud services do not support partial reads on data objects, nullifying its intended efficiency optimizations. Hence, we select RS and CRS codes for our evaluation, using *liberasurecode* to evaluate multiple backend ECC implementations of these.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Design

In this chapter, we provide an overview of the requirements and design for a proof of retrievability library supporting recovery. We review strategies for managing metadata and parity data for the library.

4.1 Interfaces and Requirements

Our POR library should implement all those interfaces required for applications seeking to implement POR challenge/response protocols. Following the POR model described by Shacham and Waters [6], we define a proof of retrievability scheme in the symmetric setting using the following interfaces:

- POR_KeyGen(1^κ), the function used to generate the keys K employed by the scheme.
- POR_Encode(K, F), the function encoding the original file F using the adversarial erasure code, to generate the encoded file $\widetilde{F}$.
- POR_Tag($K, \widetilde{F}$), the function generating the metadata η and tags σ for $\widetilde{F}$.
- POR_Challenge(K, η), the function generating a challenge c to be sent to the prover.
- POR_Proof($c, \sigma, \widetilde{F}$), the function generating the proof γ as response to challenge c.
- POR_Verify(K, c, γ), the function verifying the correctness of the proof γ.
- POR_Recover($K, \widetilde{F}'$), the function that attempts to recover the file F from some, possibly incomplete, version of $\widetilde{F}$.

Furthermore, our POR library design must be flexible enough to support a variety of symmetric-key and public-key POR schemes. The library should support different erasure codes. Also, it should be modular and support multiple error-correcting code backend libraries, i.e., different implementations of those codes.

4.2 Data and Memory Utilization

The file F may be arbitrarily large and, thus, may not fit in working memory during the encode and tag processes. Prior performance studies demonstrate that care must be taken when utilizing ECC and selecting parameters such as chunk size and code word size. Thus,

our design sub-divides F into L logical blocks of size ℓ, processed independently. Each logical block will be considered as k data blocks and expanded into m parity blocks. How to organize these blocks without further, and unnecessarily, increasing the remote data storage requires discussion. Abstractly, we consider three general approaches to managing these chunks:

1. Store the data blocks and parity blocks in separate, logical, remote files (see Figure 4.1).
2. Interleave data and parity blocks in a single, logical, remote file (see Figure 4.2).
3. Store the data blocks contiguously, appending all parity blocks, to form a single, logical, remote file (see Figure 4.3).

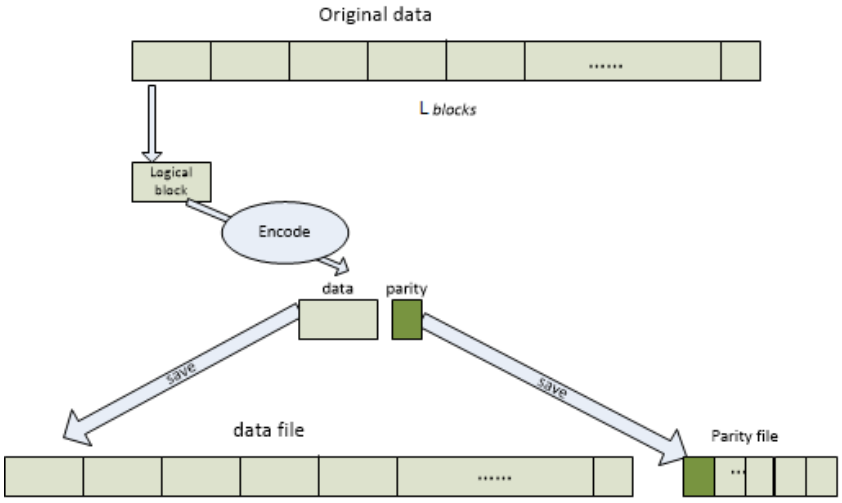


Figure 4.1: ECC block storage design (Option 1).

Option 1 provides the user with "natural" access to the original file when there are no failures, as the data file can be made identical to F. This removes the need to decode on every access. In the presence of errors, recovery can be achieved by reading incrementally from both files. To encode, we read ℓ bytes from F and encode this as k data blocks of m parity blocks, each of size ℓ/k; we write each block, sequentially, to its respective file. To decode, we read ℓ bytes from the data file and $m\ell/k$ bytes from the parity file. We need to repeat the procedure until every logical block is recovered, combing these to recover F.

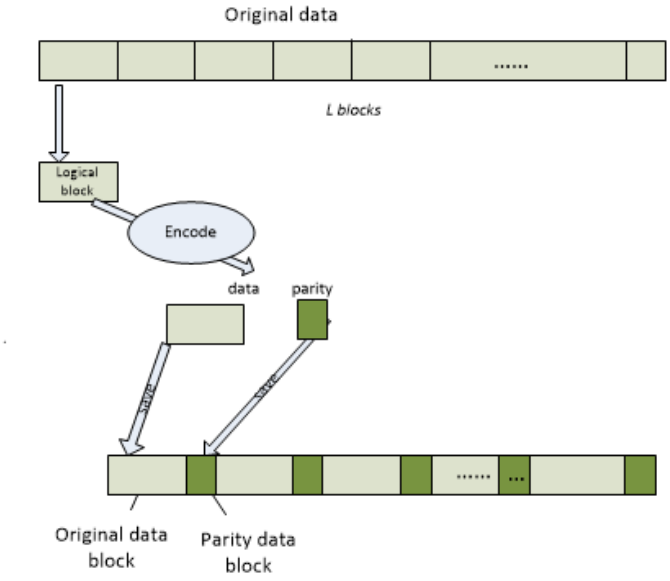


Figure 4.2: ECC block storage design (Option 2).

Option 2 is similar to the first option procedurally, but data and parity blocks are stored interleaved as a single logical file. This storage makes regular access inconvenient because it does not allow "natural" access to the original file data F.

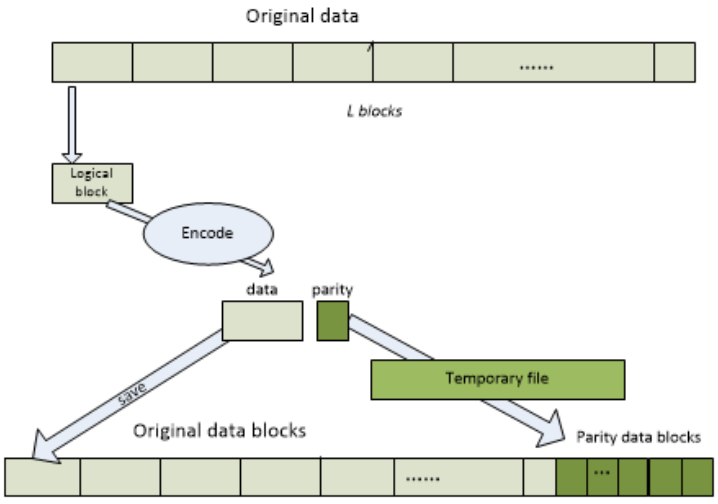


Figure 4.3: ECC block storage design (Option 3).

Option 3 stores data and parity in a single file, but stores all parity data at the end. This allows "regular" access to the file data through accessing the first $|F|$ bytes of the encoded file's prefix. Procedurally, however, encoding requires more memory or intermediate storage to hold parity blocks and write these at the end. Also, decoding requires scanning the encoded file to retrieve those parity blocks associated with the data blocks to recover each logical block.

Option 1 is most convenient for client access but inconvenient for audit, since most POR literature describes how to audit a single file, rather than two files that must be treated as a single, logical file. Comparatively, Option 2 and Option 3 yield simpler audit logic. Of these options, Option 2 has simpler encode and decode logic. We select Option 2 for our design, while noting that Options 1 and 3 have preferable properties with respect to file access in the absence of errors. We defer exploring these options more completely to future work.

CHAPTER 5:
Implementation and Analysis

In this chapter, we describe our proof of retrievability implementation employing erasure coding. Then, we discuss and analyze the performance of this implementation for those metrics of relevance to real-world cost.

5.1 Implementation

Our implementation is an extension of the open source library *libpdp*, a library supporting proof of possession written in C that currently does not include the capability to restore data under partial erasure [40]. As discussed in Chapter 3, there are many open source libraries implementing erasure codes. Our primary solution is to extend *libpdp* to support recoverability by employing an appropriate erasure code library. The current *libpdp* library provides the following five interfaces:

- pdp_key_gen: generates keys needed by the scheme;
- pdp_tags_gen: generates the tag data related to the file to be stored in the server;
- $pdp\_challenge\_gen$: generates the challenge to be sent to the Prover;
- pdp_proof_gen: generates the proof to answer the challenge;
- pdp_proof_verify: verifies the given proof.

The *libpdp* library currently supports the following four PDP schemes:

- **MACPDP**: the simple MAC-based PDP scheme;
- **APDP**: the Ateniese et al. PDP scheme [7];
- **CPOR**: the Shacham and Waters POR scheme, implemented as a PDP scheme [6];
- **SEPDP**: the Ateniese et al. PDP scheme [41].

We extend *libpdp* with new interfaces, relying on an existing backend erasure code library. We adopt the *liberasurecode* library, which not only implements some erasure codes but also interfaces with existing ECC libraries. The *liberasurecode* library consists of three main interfaces:

- *liberasurecode_encode*: generates the erasure encoded data;
- *liberasurecode_decode*: reconstructs the original data from a set of at least k encoded fragments;
- *liberasurecode_reconstruct_fragment*: reconstructs a missing fragment from a subset of available fragments.

The function *liberasurecode_encode* transforms a logical blocks into k data fragments and m parity fragments. Figure 5.1 summarizes this behavior. As part of encoding, this library function adds an 80 byte header to each encoded fragment, resulting in $80n$ bytes of additional information. The fragments will be serialized and stored as previously described (see Chapter 4, Option 2). We reduce the storage overhead associated with encoding data by removing these headers during storage and reconstructing them before decoding.

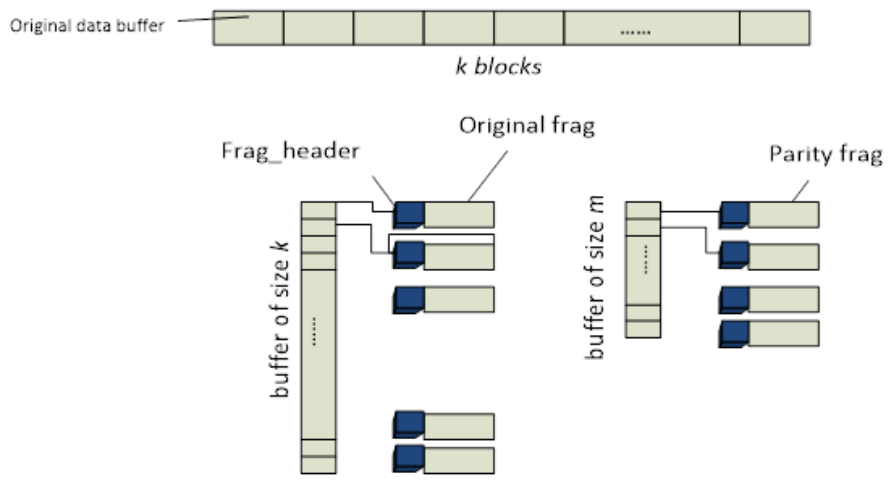


Figure 5.1: Encoding with *liberasurecode*.

We introduce two new interfaces for the *libpdp* library:

- *pdp_preprocess*: encodes the original file;
- *pdp_recover*: reconstructs the original file from k blocks of the encoded file.

5.2 Experiment Design

Our goal is to provide a real-world cost analysis of using erasure codes to implement POR schemes. Bremer provides a cost analysis of the PDP schemes available in *libpdp* [1]. We analyze our POR implementation using similar metrics and methodology to compare with Bremer's prior study.

Plank et al. provides a performance evaluation of select MDS code implementations [18]. In part, their goal is to determine the (k, m, w) parameters that ensure optimal storage overhead, best fault tolerance, and fastest encoding/decoding rate. While we aim for many of the same goals, we re-evaluate their results to establish their applicability to our setting. In particular, their primary application is redundant array of independent disks storage, and our cloud storage setting is quite different. Also, the libraries we explore are different and more recent than this in earlier study, called via the *liberasurecode* wrapper library. Otherwise, we attempt to follow a comparable procedure to that of Plank et al., to determine good ECC parameters to be employed by our POR library. We study four MDS codes implementations:

- Reed-Solomon implemented by Jerasure, denoted JRS.
- Cauchy Reed-Solomon implemented by Jerasure, denoted JRC.
- Reed-Solomon implemented by liberasurecode, denoted LER.
- Reed-Solomon implemented by ISA-Intel, denoted ISA.

We explore these using three (k, m) combinations and varying the value of w from 4 to 32, following the parameter space explored by Plank et al.

Finally, we test the performance of the available POR schemes with file pre-processing using an MDS code with (k, m, w) parameters that appear to perform well. This allows us to interpret the real-world cost of POR schemes in the context of Bremer's prior work [1].

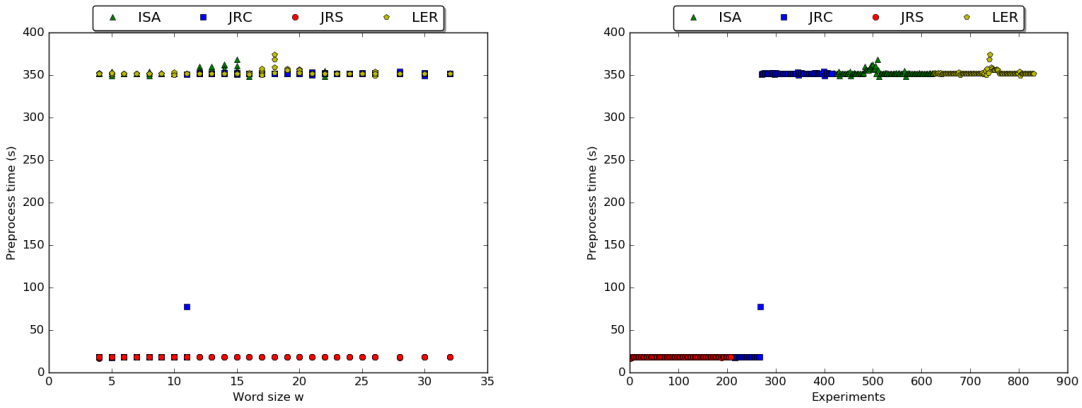
5.2.1 Experimental Environment

Our test environment is a Dell laptop running Windows 7 enterprise with a 64-bit 2.7GHz Intel Core i7, with 8GB of RAM. The host is running a VMware version 7.0.0. The virtual machine is 64-bit Ubuntu 14.04-LTS assigned 3GB memory. Actually, this environment has more memory and faster processing capabilities than Plank's test environment. These

capabilities allow us to easily run the virtual machine and get reasonable results in an acceptable amount of time.

5.2.2 Cloud Environment

As POR is designed to audit cloud storage, the most natural environment to evaluate our library is a cloud environment. Our initial experimentation used a virtual machine running on Amazon Web Services (AWS): an Amazon c3.xlarge instance running 64-bit, Ubuntu 14.04-LTS with 8GB memory and using Hardware-assisted Virtual Machine (HVM) virtualization. These preliminary tests revealed strange I/O behavior (see Figure 5.2). After a short period of time, all I/O operations suffer a dramatic loss of performance. This behavior is demonstrated in some form, no matter the order of the tests, file size or parameters. We did not observe this behavior using our desktop virtual test environment. We defer further study of this I/O behavior on Amazon instances to future work.



(a) Word size vs. preprocessing time

(b) Test order vs. preprocessing time

Figure 5.2: Preprocessing performance for k=6 and m=2 on Amazon c3.xlarge, showing performance sensitivity related to test order.

5.3 Analysis

As previously outlined, our first set of experiments is intended to remake Plank [18] open source erasure codes libraries performances. Then, we test the preprocessing time to characterize the encoding speed compared to the file size. After, we study the tag cost

that could reveal the impact of introducing erasure code to four POR schemes. Finally, we measured the storage overhead resulted by the addition of the erasure code and compared to prior Bremer [1] study.

5.3.1 Encoding Performance

We investigate three (k, m) combinations—$(6, 2)$, $(12, 4)$ and $(14, 2)$— each time employing a randomly generated 1 GB file. We vary word size w from 4 to 32 for all schemes; however, per its documentation, JRS only meaningfully handles values $w = 8, 16, 32$. Each trial is repeated five times. We investigate using two logical block sizes: 100 KB (as explored by Plank et al.), and 64 MB (a common maximum chunk size for some distributed cloud file systems). We instrumented the *liberasurecode_encode* function to perform measurement, avoiding calls like *read* and *write* that incur I/O. This was an attempt to measure encoding speed more closely following prior work of Plank et al., which excluded I/O costs.

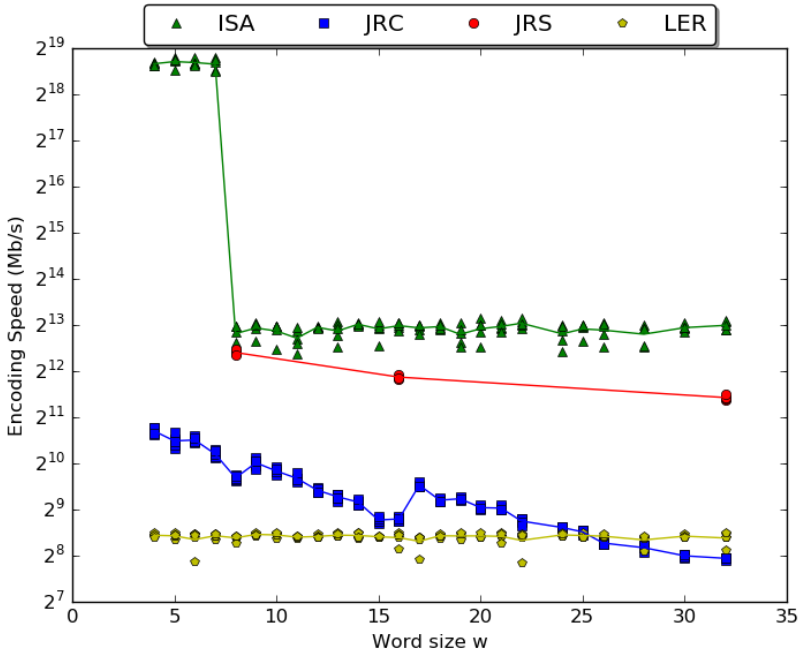


Figure 5.3: Word size vs. encoding speed for k=6, m=2, ℓ=100KB.

Figures 5.3, 5.4 and 5.5 summarize the results of these experiments using 100 KB logical block size. There are significant differences compared to the prior performance analysis of

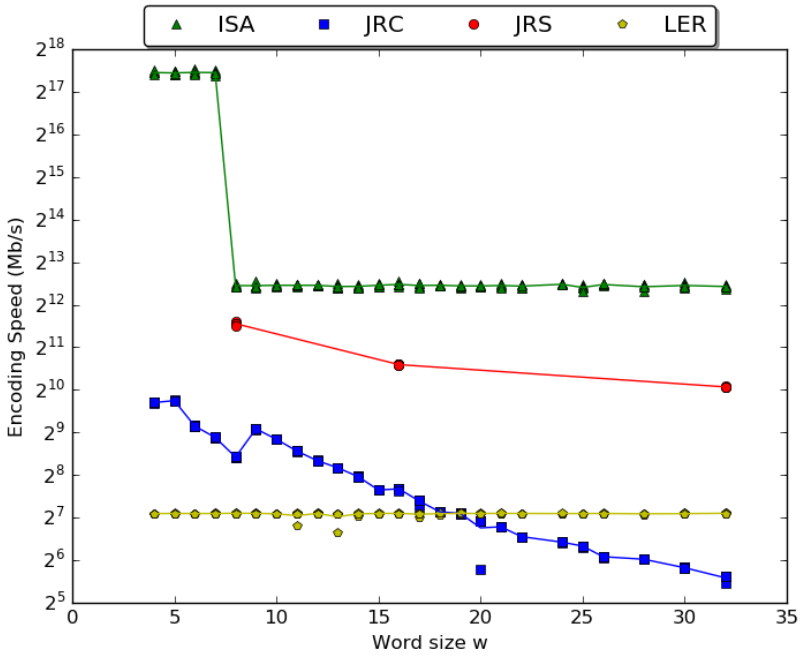


Figure 5.4: Word size vs. encoding speed for k=12, m=4, ℓ=100KB.

Plank et al. The first is that our test environment demonstrates, overall, a much higher encoding speed for all schemes evaluated. We expected, however, to see relatively comparable performance patterns. For example, we expected JRC to show better performance than JRS across all word sized; however, we observed no such patter. Also, compared to Plank et al., we did not observe the same relationship between the word size and the CRS encoding performance. There are no specific word sizes w resulting in worse performance. But, we do notice that CRS performance decrease when the word size increase. In fact, all schemes present better encoding performance with small word size. For instance, RS implemented by ISA-Intel present the fastest encoding speed and it has better performance for $w < 8$.

Figures 5.3, 5.4 and 5.5 summarize the results of these experiments using 64MB logical block size. They do not present any significant deviation from the behavior observed with 100KB logical block size. Overall, we observe encoding speed is faster using a larger logical block size and small word size. Furthermore, ISA-Intel implements the fastest encoding performance among the four tested schemes. We expected to observe that Cauchy Reed-Solomon would be faster than Reed-Solomon for all experiments, but the experimental results did not comply with this intuition. There are a number of possible explanations for the

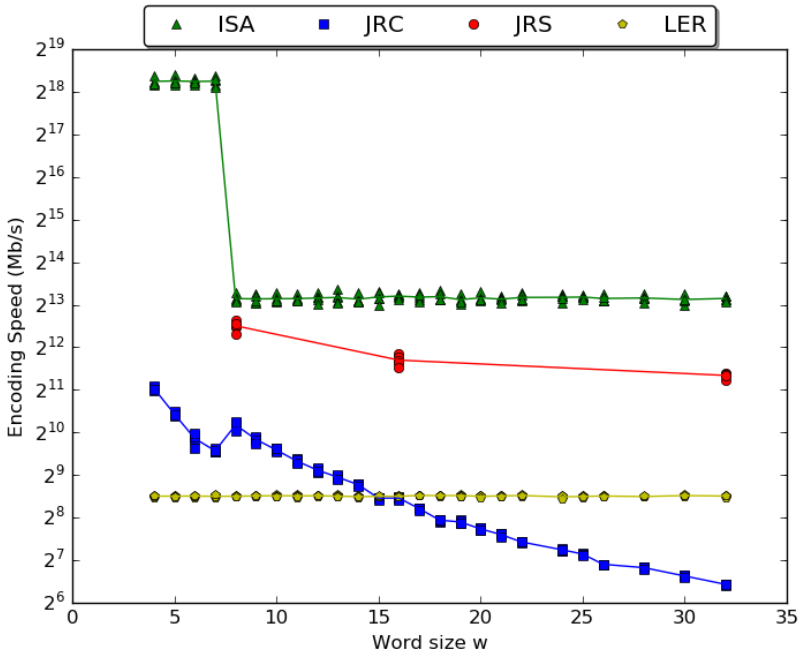


Figure 5.5: Word size vs. encoding speed for k=14, m=2, ℓ=100KB.

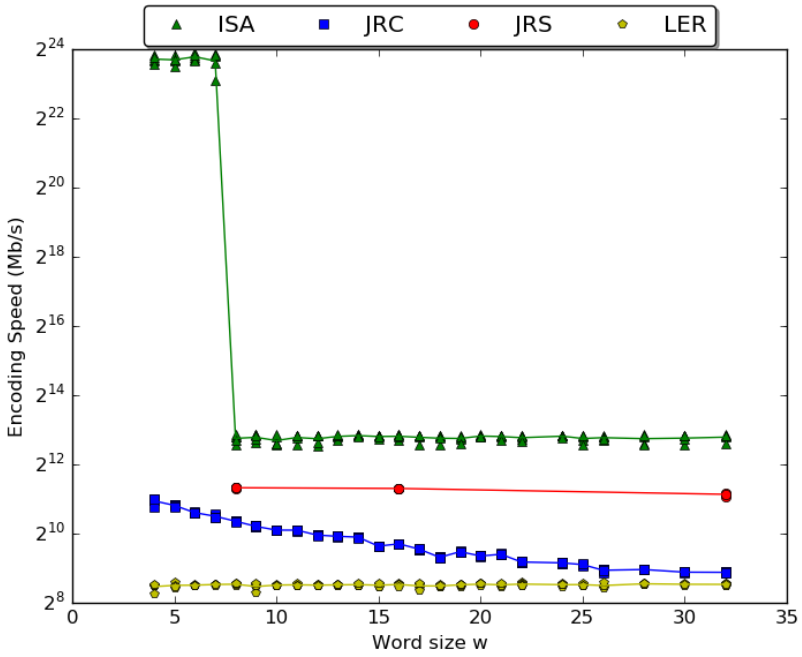


Figure 5.6: Word size vs. encoding speed for k=6, m=2, ℓ=64MB.

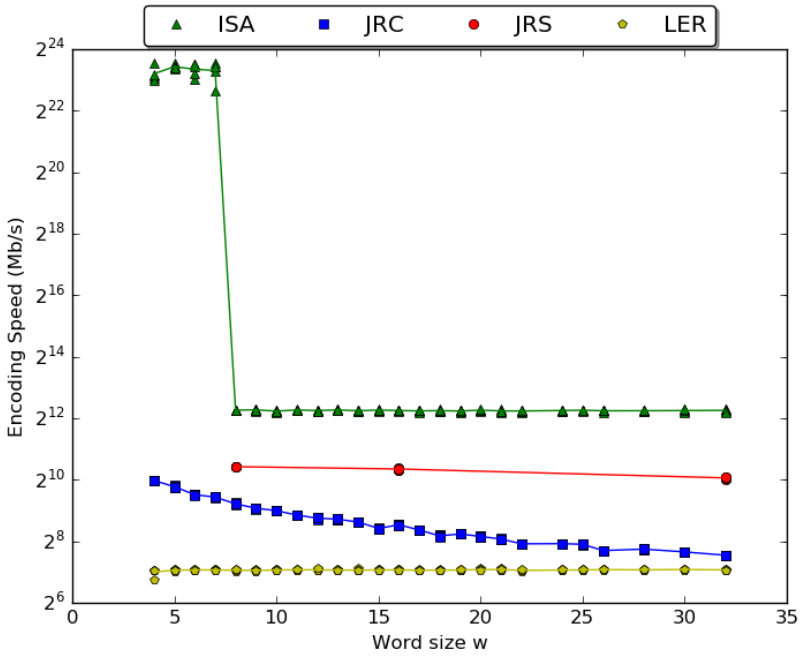


Figure 5.7: Word size vs. encoding speed for k=12, m=4, ℓ=64MB.

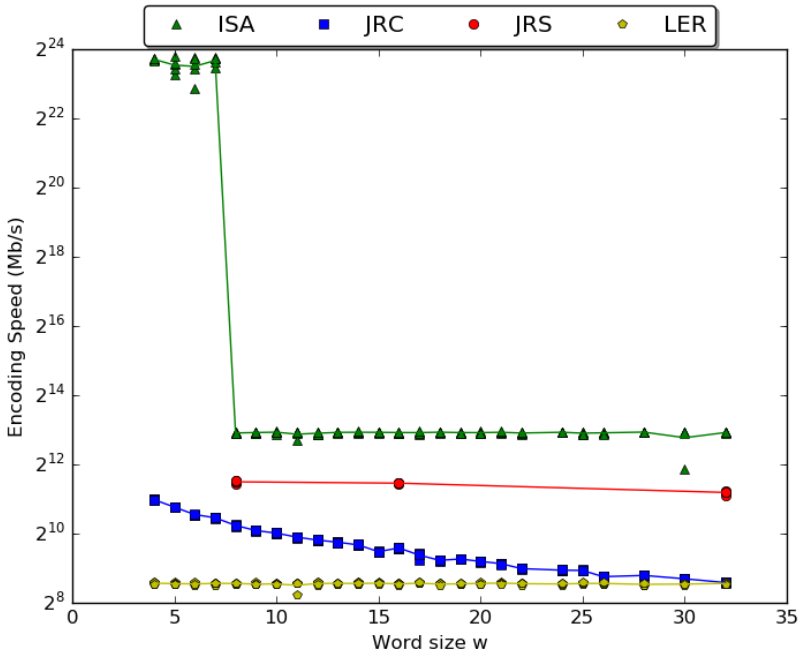
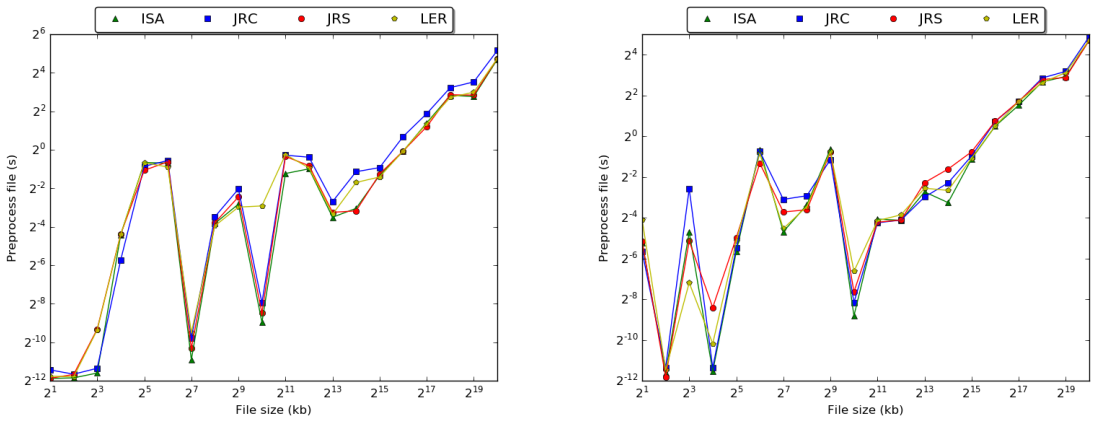


Figure 5.8: Word size vs. encoding speed for k=14, m=2, ℓ=64MB.

differences between the observations of Plank et al. and our observations: our experimental environment is different, our method of instrumenting measurement in the backend library may have failed to exclude some I/O costs, the *liberasurecode* library may have introduced some overhead that undermines the relative efficiency of the CRS backend implementation, or our experiment environment may have introduced some limiting factor that constrained performance in some unanticipated way. The result is that this preliminary performance experimentation shows that the encoding schemes present better encoding speed when small word size w is adopted. Thus, for the following evaluation, we will use the encoding schemes with the default k and m values given by the developer of *Liberasurecode* library and with word size w=8 for JRS and w=4 for the other schemes.

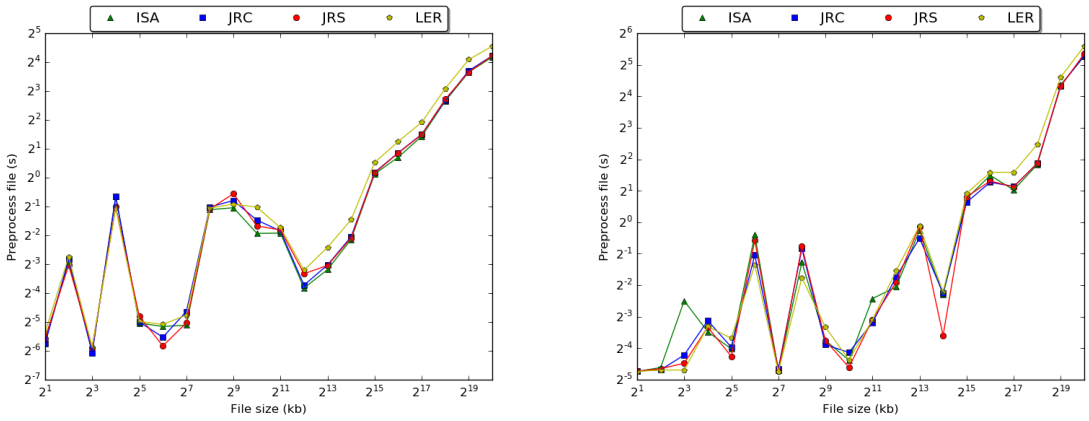
5.3.2 Pre-processing Performance

Pre-processing is the procedure encoding the file F and outputting the file $\widetilde{F}$. This operation includes file encoding and related I/O operations. Thus, the logical block size has direct effect on the total pre-processing time via the cost of *read* and *write* operations. We test four different logical block sizes: 50KB, 100KB, 64MB and 120MB. We generate random files of size 2kB to 1 GB and we run our tests 10 times for each of the four schemes evaluated (plotted results are averages). Figures 5.9 and 5.10 summarize the results of these experiments.



(a) 50KB logical block size (b) 100KB logical block size

Figure 5.9: File size vs. preprocessing time for ℓ=50KB and ℓ=100KB.



(a) 64MB logical block size

(b) 128Mb logical block size

Figure 5.10: File size vs. preprocessing time for ℓ=64MB and ℓ=128MB.

As expected, for large files sizes, we see a nearly linear relationship between file size and preprocessing time. For small file sizes, we observe surprisingly non-proportional performance we believe related to I/O costs or other activity interfering with our measurement.

5.3.3 Tagging Performance

The tagging performance is measured as the total time to generate tags associated with the encoded file $\widetilde{F}$. We generate random files of size 2kB to 1 GB and we run our tests 10 times for each of the four schemes evaluated (plotted results are averages). In this test, we used only RS implemented by Jerasure with the parameters k=10 and m=4 as the erasure code. Figure 5.11 summarizes the performance of generating tags using the POR scheme parameters described by Bremer, evaluated using our prototype library.

As expected, the use of erasure coding increases the time to tag the encoded file relative to the time to tag the unencoded file. The trends we observe for each scheme are identical compared to those previously reported by Bremer [1]. The tagging costs of the encoded file are a simple linear shift of those costs for unencoded files.

The ratio m/k is the primary factor affecting this tag time. By increasing this ratio, we increase the size of the file to tag, which will increase the tag time proportionally. For the SEPDP scheme, the tag time increases linearly up to a point, after which the tag time remains constant, when the file size exceeds the fixed number of tags to be generated; these

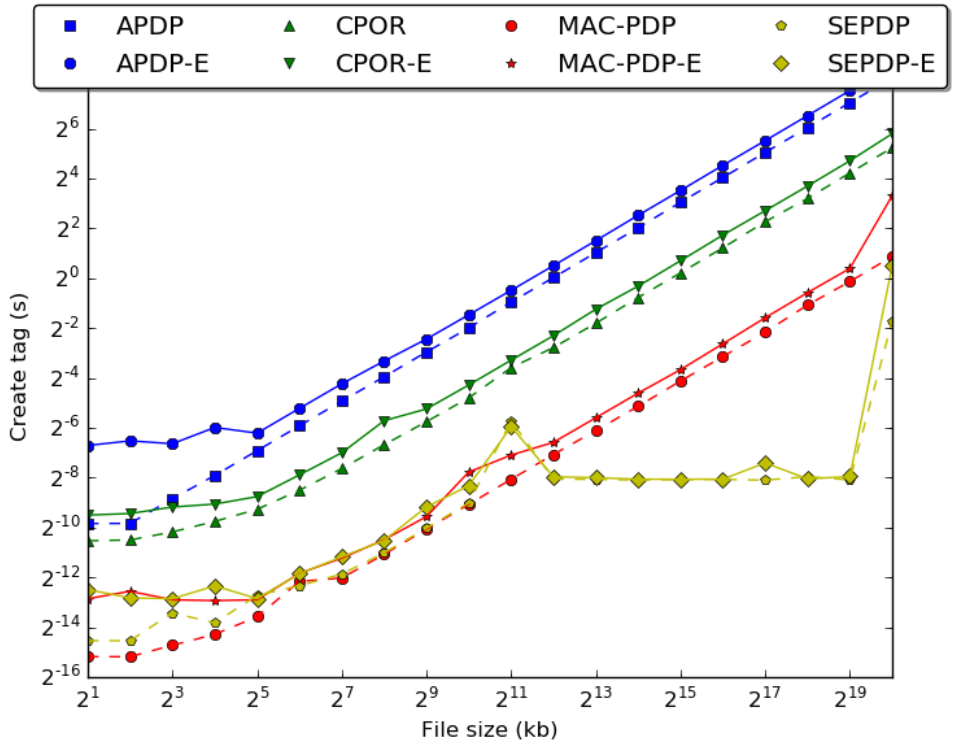


Figure 5.11: File size vs. POR tag time.

same trends for SEPDP are observed by Bremer.

5.3.4 Storage overhead

The storage overhead for F is the sum of the parity and the tag data. We measure the overhead for three of our POR schemes, excluding SEPDP. Table 5.1 shows the total storage overhead expected based on prior analysis. Figure 5.12 expresses the measured overheads experienced by our library, which match those expected closely. The results largely reflect the addition of m parity blocks to the encoded file storage.

Table 5.1: File storage overhead for each POR scheme (bs = 4096 bytes).

Scheme	Tag size (bytes)	Tag file overhead (% $\lvert\widetilde{F}\rvert$)	Total storage overhead with ECC (% $\lvert F\rvert$)
A-PDP	204	4.864%	$m/k + 4.864\%(1 + m/k)$
MAC-PDP	20	0.477%	$m/k + 0.477\%(1 + m/k)$
CPOR	18	0.429%	$m/k + 0.429\%(1 + m/k)$

Adapted from [1]: S. J. Bremer, "Cost comparison among provable data possession schemes," M.S. thesis, Dept. Comput. Sci., Naval Postgraduate School (NPS), Monterey, 2016.

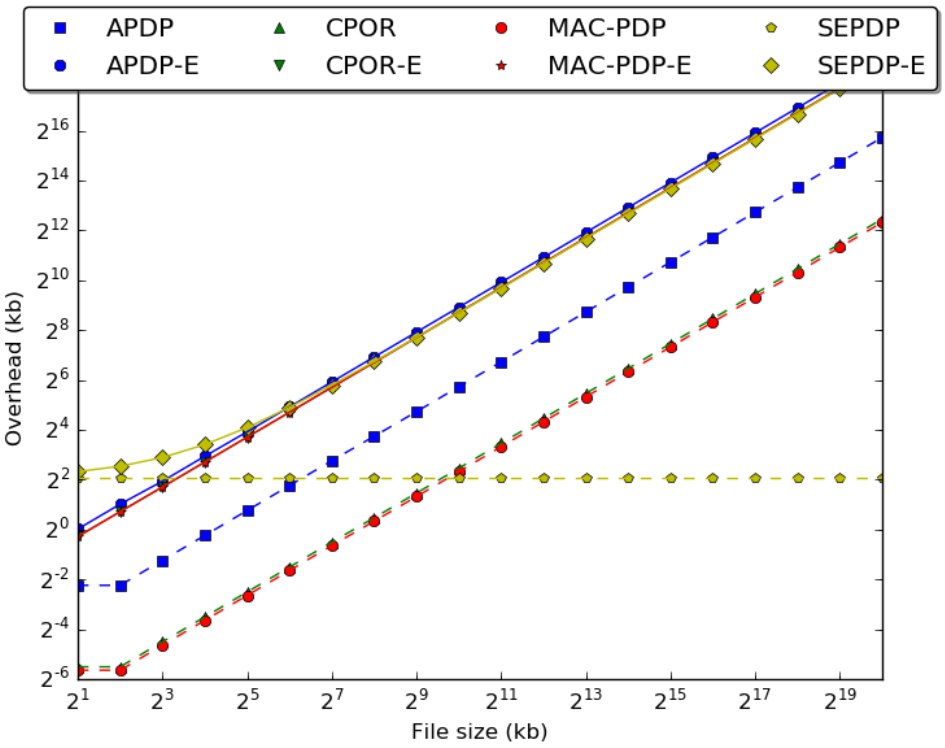


Figure 5.12: File size vs. POR overhead using JRS with k=10 and m=4.

CHAPTER 6:
Conclusion

In this work, we analyzed the maximum distance separable codes as adversarial erasure codes to retrieve data stored in the cloud. Then we developed and added a retrievability interface to the PDP library, $libpdp$. Furthermore, we developed generic cost models for two erasure codes implemented by three different backend libraries. We analyzed the selection of parameters (k, m, w) on the performance of the erasure codes. Following that analysis, we studied the cost impact of the use of erasure codes on tagging files. Our results showed that using the erasure code adds additional tag cost. Then, we analyzed the storage overhead of three POR schemes. Our results showed that the integration of the erasure code will increase storage cost proportional to m/k.

While our study covered only a static retrievability of data saved on a single server, future studies could incorporate a real-world cost of data updates on the performance of proof of retrievability. Also, we studied only one option of memory utilization in our library. Follow-on work could implement and study other options that could provide a more useful solution. In Chapter 5, we observed that cloud-based experimentation with Amazon instances introduces some unusual input/output behavior. Future work could investigate this issue, perhaps selecting an Amazon instance providing better input/output (I/O) performance for the purposes of stress-testing the encode and decode operations. Also, it could investigate the anomalies we say with respect to the relative speed of JRS and JRC, which did not reflect prior observations of Plank et al.

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] S. J. Bremer, "Cost comparison among provable data possession schemes," M.S. thesis, Dept. Comput. Sci., Naval Postgraduate School (NPS), Monterey, 2016.

[2] J. Gantz and D. Reinsel. (2012). THE DIGITAL UNIVERSE IN 2020: Big Data,Bigger Digital Shadows, and Biggest Grow th in the Far East. [Online]. Available: https://www.emc.com/collateral/analyst-reports/idc-digital-universe-united-states.pdf

[3] Vormetic. (2015). 2015 Vormetic InsiderThreat Report. [Online]. Available: http://enterprise-encryption.vormetric.com/rs/vormetric/images/CW_GlobalReport_2015_Insider_threat_Vormetric_Single_Pages_010915.pdf

[4] A. Juels and B. Kaliski, "PORs: Proofs of retrievability for large files," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, 2007, pp. 584–597.

[5] M. Naor and G. Rothblum, "The complexity of online memory checking," in *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, 2005, pp. 573–584.

[6] H. Shacham and B. Waters, "Compact proofs of retrievability," in *Proceedings of The 14th Annual International Conference on the Theory and Application of Cryptology and Information Security*. ASIACRYPT, 2008, pp. 90–107.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Scalable and efficient provable data possession," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS 2007)*, 2007, p. 598.

[8] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "MR-PDP: Multiple-replica provable data possession," in *Proceedings of the 28th International Conference on Distributed Computing Systems*. ASIACRYPT, 2008, pp. 90–107.

[9] B. Chen, R. Curtmola, G. Ateniese, and R. Burns, "Remote data checking for network coding-based distributed storage systems," in *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, 2010, pp. 31–42.

[10] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A high-availability and integrity layer for cloud storage," in *Proceedings of 16th ACM Conference on Computer and Communications Security*, 2009, pp. 187–189.

[11] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, pp. 379–423, 1948.

[12] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, pp. 147–160, 1950.

[13] J. S. Plank and C. Huang, "Tutorial: Erasure coding for storage applications," Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, San Jose, February 2013.

[14] R. J. Lipton, "A new approach to information theory," in *Proceedings of 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, 1994, pp. 699–708.

[15] V. Guruswami and A. Smith, "Codes for computationally simple channels: Explicit constructions with optimal rate," in *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, 2010, p. 723–732.

[16] J. S. Plank, "XOR's, lower bounds and mds codes for storage," in *Proceedings of the IEEE Information Theory Workshop*, 2011, pp. 503–507.

[17] K. Bowers, A. Juels, and A. Oprea, "Proofs of retrievability: Theory and implementation," *Cryptology ePrint Archive, Rep. 175*, 2008.

[18] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proceedings of the 7th Conference on File and Storage Technologies(FAST)*, 2009, pp. 253–265.

[19] M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2013, pp. 66–75.

[20] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial & Applied Mathematics*, 1960.

[21] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," *International Computer Science Institute Report. TR-95-048*, 1995.

[22] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: an optimal scheme for tolerating double disk failures in raid architectures," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*. ISCA, 1994, pp. 245–254.

[23] M. Blaum and R. M. Roth, "On lowest density mds codes," in *Proceedings of the IEEE Transactions on Information Theory*, 1999, pp. 46–59.

[24] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing i/o for recovery and degraded reads," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012, p. 251.

[25] M. Schnjakin, T. Metzke, and C. Meinel, "Applying erasure codes for fault tolerance in cloud-raid," in *Proceedings of the IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2010, pp. 66–75.

[26] G. Tucker. (2014, Dec.). Intel Storage Acceleration Library (Open Source Version). [Online]. Available: http://goo.gl/ZNtMkZ

[27] B. Beach. (2015). JavaReedSolomon. [Online]. Available: https://github.com/Backblaze/JavaReedSolomon

[28] S. Plank, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications," *CS-07-603, University of Tennessee*, 2007.

[29] Steinwurf, "Kodo-c 8.0.1," Jan. 2016. Available: https://github.com/steinwurf/kodo-c

[30] T. Gohad. (2016). liberasurecode. [Online]. Available: https://bitbucket.org/tsg-/liberasurecode/

[31] INREA, "Openfec 1.4.2," Dec. 2014. Available: http://openfec.org/downloads.html

[32] K. Greenan and T. Gohad. (2016). PyECLib 1.2.0. [Online]. Available: https://pypi.python.org/pypi/PyECLib

[33] Z. Wilcox-O'Hearn. (2012). Zfec 1.4.24. [Online]. Available: https://pypi.python.org/pypi/zfec/1.4.24

[34] P. Luse and K. Greenan. (2014, Nov.). Taking the Mystery out of Erasure Codes: A Swift Implementation. [Online]. Available: https://01.org/sites/default/files/documentation/swiftec-kilo-summit-final.pdf

[35] F. Vennetier and J.-F. Smigielski. (2014). librain. [Online]. Available: https://github.com/open-io/librain

[36] Z. Zhang, A. Wang, K. Zheng, U. M. G, and V. B, "Introduction to HDFS erasure coding in Apache Hadoop," 2015. Available: https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

[37] "Hdfs-7285: Erasure coding support inside hdfs," Sep. 2015. Available: https://issues.apache.org/jira/browse/HDFS-7285

[38] "HDFS-RAID," Nov. 2011. Available: https://wiki.apache.org/hadoop/HDFS-RAID

[39] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proceedings 7th Usenix Conference on File and Storage Technologies (FAST)*, 2009.

[40] M. Gondree, "libpdp." Available: https://github.com/gondree/libpdp

[41] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik, "Scalable and efficient provable data possession," in *Proceedings of the 4th International Conference on Security and Privacy in Communication Networks*, 2008, p. 9.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California